The Sourceror's Apprentice

The Assembly Language Journal of Merlin Programmers Vol 1 No 10 October 1989

Mike Rochip Mouths Off

Mike Rochip: A determined man

Dear Mike,

I enjoy your publication very much, however there are a few things that I would like to point out or ask about.

In your September '89 editorial you said something about "limited by space". In comparing your newsletter to ***A2-Central****, it looks like you have too much space. I would like it better if you used a much smaller typeface, smaller (or fewer) pictures, and double column assembly listings. For $28/year, I need more stuff. Are you not receiving enough material to publish? Just look at all the unused space on pages 8-11 and 16 of September '89! A whole page just for copyright info?*

I really like your editorial comments in the articles. I believe even more are in order (ala Cecil's in ***CALL A.P.P.L.E.****).*

We really need a good assembly newsletter like yours. (I really hated seeing Bob's go away.) You are improving, though maybe you tackled too many magazines at the same time.

Thank you for your time.

Robert Muir
FPO San Francisco

Dear Robert,

First, I must say that your letter had quite an effect on me. I was upset, initially, because I'm not certain most folks understand the newsletter industry and the economics thereof. But it also made me think, so I suppose that it served a good purpose. Furthermore, it has served as the springboard for change, and an excuse to share with the subscribership the constraints under which a newsletter of this nature must operate.

You'll notice the changes in the layout of this issue. They were prompted by your comments, Robert, and those of all the rest of you who have written regarding this. That is one of the advantages of a newsletter over a magazine.

However, I think you all need to understand what was going through my mind as I produced the last issue, and the costs I must balance with every issue.

Stop and consider that *The Sourceror's Apprentice* is usually advertised as a 12 page newsletter. Now consider that last month I put out a 16 page newsletter. Yes, there were big pictures and some white space. But would you have been happier had I crammed everything into 12 pages? I admit, I only had about 14 pages of material. In such a situation I am faced with either 1) going with 12 pages and serializing at least one article, or 2) spreading out

the 14 pages into 16.

Sure, I could have printed another short article. But **articles cost money** or time. Have we forgotten that Bob S-C just flat ran out of both with *AAL*? I have precious little of either commodity myself. I have plenty of articles and source code to print, but as soon as I do I owe someone some money. I have paid *every* contributor *something* for their work, although usually only $25 - $75. I am convinced that it is a) the morally correct thing to do, and b) necessary for attracting high quality articles.

So am I just too cheap to print a bigger rag? Let's look at some facts and figures:

Income: The total subscriber base is currently about 400. The average price paid has been $25 (we've had some intro offers and what-not). That means the total income from the newsletter has been about $10,000. Since the quarterly disk is barely a break-even deal, I'll exclude it from these figures.

Expenses: A 16 page issue (as most have been) costs me $289 to have 450 printed (with the quality of print and paper I demand). That's $3468 per year. Postage for a 16 pager is $.45 per issue. That works out to $2160 per year. Articles cost between $75 to $150 per month. Let's split the difference and call it $110. That's $1320 per year.

Not including office expenses, phone bills, envelopes or my time, we come out with $6948 total expenses. Let's call it $7000.

That leaves $3000 to pay for any marketing and expansion - and my salary. For an entire year.

Some folks have said that I ought to allow advertising. Fine, except that I cannot find any advertisers. The fact is, the market of hard core Apple II assembly language programmers is sufficiently narrow (probably less than 1500 total) that **nobody** is going to pay very much for advertising herein. One major Apple II hardware company told me that they wouldn't advertise here if I gave them the space for free - it wouldn't be worth their time and effort. Incidentally, this same company *did* advertise in *AAL* for awhile, so they were basing their decision on experience.

A publication like this one can *only* exist in newsletter form (i.e. subscriber supported).

I am a little sensitive to the comment about needing more for $28. I have spent a considerable amount of time over the last six months talking to various business and marketing people about Ariel Publishing. Without fail, every single person has advised me to **triple or quadruple** the prices of my publications. They point out, correctly, that highly technical information that cannot be got elsewhere is worth a lot (we were the first to discuss the new tool startups, the first with resources, and the only Apple II publication dedicated to teaching and sharing assembly language programming techniques.). Many financial newsletters, for example, routinely charge $140 or more per year. One commodities newsletter I've seen is over $900 per year! Even if I lost 50% of my subscribers, I've been told, I'd still come out way ahead.

I'm not going to do that, however. Instead, here in the short run, I'm going to cut costs by limiting each issue to 12 pages (our advertised length all along!). As compensation to y'all, the new layout will squeeze in more per page.

And consider the price of Apple's Partner Program per year, too ($600). Granted, we do not dish out as much information (or junk mail). But they do not explain things as well (I think) nor do they screen out extraneous info.

I am determined to avoid joining the long list of failed, deceased, or struggling Apple II publications (which definitely includes the now gasping *CALL A.P.P.L.E.)*. The list of which I speak is not necessarily a commentary on the state of the Apple II market, by the way, since our beloved *Softalk* died during the heyday of the II. The reasons in each case have been a combination of business blunders and an inability to keep costs down. Of course, Apple's lack of marketing momentum and money has not helped, either.

Well, I guess I have thoroughly vented my spleen here, haven't I? I hope nobody found it offensive - my intent was to inform and educate. I also hope some of you found it enlightening. If you like *The Apprentice* and would like to help insure its success, the best thing you could do would be to talk a friend into subscribing.

Frankly, I have been a little discouraged and frustrated (you could tell?) by the lack of success of my last two marketing ventures (embarrassing for a supposed "guru" of small business marketing). I fully expected to be up over 1,000 subscribers by now.

Which leads me to my final point: Robert's letter really DID make me think (and I thank you for it, Robert, and for your diplomacy). As much as I have berated Apple and others for being insensitive to their markets, I am nevertheless a little guilty of that myself.

I have therefore included a quick survey on an insert to find out what y'all are thinking and how I can make this a better newsletter, more in tune with your needs.

I'll do everything I can, but I thought you all should know what constrains us.

DesignMaster?

By the way, a couple people pointed out that I prepared y'all for the shock of the new price of DesignMaster but never delivered the shock itself. Whoops. Who proofreads this newsletter, anyway? ... The ByteWorks has placed a suggested retail price of $95 on the latest version (compatible with system software version 5.0). One thing I forgot to mention was that the program is *greatly* expanded and enhanced, thus there is yet another reason for the price increase. The new version is currently beta testing.

Macro Mania

I'd also like to place a call for macro maniacs - do I have any volunteers for the APP.BUILDER to Merlin translation? APP.BUILDER is Eric Soldan's macro masterpiece for 8 bit Apple II assembly language. I won't have the time to do the translation work for awhile. Anybody else want a whack at it? You'd be doing all Merlin 8-bitters everywhere a terrific service! Since APP.BUILDER was created by Apple, I cannot sell it, hence I cannot pay anyone to do the translation work. But I can give away free copies - if I had a Merlin version...

Wanna Be a SubContractor?

Would anybody like to go into business for themselves as our quarterly disk distributor? I'd send you the contents of the newsletter each month on disk (plus other goodies). Your job would be to arrange them into a package, copy the disks, and mail out them out. The most onerous duty of it all is the disk duplication (yuk). We have about 85 disk subscribers. I'd pay a king's ransom to get this off my back. Send me a bid if yer' interested (include the costs of postage and disks).

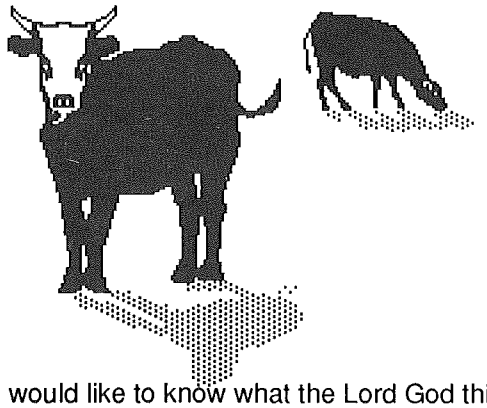
Speaking of the quarterly disk, if you are a shareware author and would like some quickie distribution, or if you have a demo version of a full fledged commercial product, we'd be overjoyed to put those buggers on our quarterly disk.

Important NEWS FLASH!!!

Ariel Publishing has a new phone number. We moved to bigger digs and "Boondocks Bell" could not let us keep our old number. The new number is (509) 923-2249.

Now back to our regularly scheduled publication... I hope you en**joy** it (a little pun considering that two of our feature articles this month have to do with joystick programming).

== Ross ==



"If you would like to know what the Lord God thinks of MS-DOS, you have only to look at those to whom he has given it."

(with apologies to Maurice Baring and Guy Kawasaki)

Just when you thought it was safe...

Honing Hot High-Res Code

by Jerry Kindall, Contributing Editor

I really enjoyed Eric Soldan's piece in the September, 1989 *Sourceror's Apprentice*. I'd never even thought it possible to get that much resolution out of a joystick. Now, though, I'm using a variation of Eric's routine in a new Apple II CAD program that Kitchen Sink Software is working on.

I feel quite odd about presuming to improve on a literal gem of a routine, but there are, indeed, a few improvements that one can make. For reference, here's the heart of Eric's original routine, sans comments, with a few equates added, and with a label attached to the first instruction of the routine, but otherwise as printed on page 3:

```
 1  ptrig    =    $C070
 2  paddl0   =    $C064
 3  yval     =    $00
 4
 5  rdpdl0   bit  ptrig
 6           lda  #$00
 7           tay
 8           clc
 9  loop     adc  #$01
10           ldx  paddl0
11           bpl  done
12           iny
13           ldx  paddl0
14           bmi  loop
15  done     sty  yval
16           adc  yval
17           ldy  #$00
18           bcc  rts0
19           iny
20  rts0     rts
```

The first thing I noticed was that the routine starts counting at 1, not 0 as we programmers are usually accustomed to counting. This happens because the accumulator starts out at zero (line 6), and is incremented (line 9) before the termination condition is even tested (line 10). This isn't necessarily a bug, but I needed to get rid of the extra count. I originally tried initializing the accumulator to $FF instead, but that caused the accumulator to jump from $FF to 0 to 2, skipping 1 entirely, because of the carry generated.

It's easier to adjust things after the fact. I stuck in a DEY right before line 15 (and gave the new line the label DONE), reasoning that it didn't matter which counter I subtracted the excess count from. However, this caused 0 to return as 256, because of the carry generated by adding 1 and 255 (the result of decrementing 0 in the Y register.) Eventually, I succumbed to being non-clever and inserted a SEC, SBC #1, CLC right before line 16. The revised code, then, follows:

```
 1  ptrig    =    $C070
 2  paddl0   =    $C064
 3  yval     =    $00
 4
 5  rdpdl0   bit  ptrig
 6           lda  #$00
 7           tay
 8           clc
 9  loop     adc  #$01
10           ldx  paddl0
11           bpl  done
12           iny
13           ldx  paddl0
14           bmi  loop
15  done     sty  yval
16           sec
17           sbc  #1
18           clc
19           adc  yval
20           ldy  #$00
21           bcc  rts0
22           iny
23  rts0     rts
```

I then got to wondering why Eric used the accumulator to count one half of the loop rather than using the X register. Using the X register would allow the INX opcode to be used in place of the ADC, shaving a cycle off the increment time. Looking at the cycle times in the comments for the listing, I realized that Eric used the accumulator and ADC to make both sides of the loop run with the same number of cycles. I wondered what would happen, though, if I did use the X register instead. The result was mind-blowing: I was able to read values of up to 410 from my joystick! Despite the minor inaccuracy introduced by the "lospidedness" of the new loop, it seemed to work fine. Here's the routine:

```
 1  ptrig   =    $C070
 2  paddl0  =    $C064
 3  yval    =    $00
 4
 5  rdpdl0  bit  ptrig
 6          ldx  #$FF
 7          ldy  #$00
 8          clc
 9  loop    inx
10          lda  paddl0
11          bpl  done
12          iny
13          lda  paddl0
14          bmi  loop
15  done    sty  yval
16          txa
17          adc  yval
18          ldy  #$00
19          bcc  rts0
20          iny
21  rts0    rts
```

I start the X register at $FF, which means that the first wrap will take me to 0; thus, there's no need to adjust the count at the end of the loop to make it start at zero. Since INX doesn't affect the carry flag, the problem I ran into with Eric's ADC loop doesn't happen here when X wraps around to zero.

JOYSTICK CONNECTIONS

Eric suggested making sure that a joystick was connected before calling his routine. Here's a routine that'll do that, returning with a set carry if there's no joystick connected and a clear carry otherwise:

```
 1  paddl0  =    $C064
 2  pread   =    $FB1E
 3
 4  chkstk  ldx  #0
 5          jsr  pread
 6          iny
 7          bne  found
 8          ldy  #2
 9          ldx  #$80
10  wait    lda  paddl0
11          dex
12          bne  wait
13          dey
14          bne  wait
15          bit  paddl0
16          bpl  found
17          sec
18          rts
19  found   clc
20          rts
```

The routine reads paddle 0, using the usual PREAD routine at $FB1E, and if any value other than 255 is found, a joystick is known to be connected. If a 255 is found, it could mean that the stick is full right, or that there's no stick at all. Lines 8-14 form a short delay routine. After the delay, if the PADDL0 timer is still high, it means that no stick at all is connected. In Kitchen Sink's CAD program, the Y coordinate needed to have a range of only 0 to 212, so I was able to use PREAD to read the Y axis, and check for a disconnected joystick just before reading the X coordinate with Eric's method, all within the same routine.

TROUBLE-FREE DELAYS

In my delay loop above, I access the paddl0 timer repeatedly. This keeps most accelerators running at 1 Mhz through the loop — a favorite trick of mine for acclerator-independent timing. This doesn't keep the IIgs going slow, though; you need to explicitly slow it down when accessing the Apple game paddle softswitches. I wrote a couple of simple routines which manipulate the IIgs speed; you might want to have the paddle routines call SLOWGS before doing anything else, and exit with a JMP to FASTGS rather than with an RTS. FASTGS doesn't actually set the GS's speed to fast, but to the speed in effect when SLOWGS was called. Here's the code:

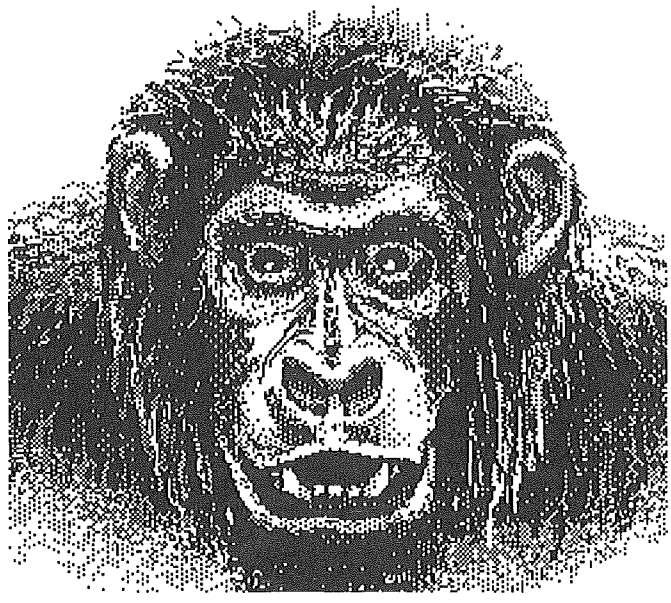
```
 1  romid    =     $FE1F
 2  speed    =     $C036
 3
 4  slowgs   sec
 5           jsr   $FE1F
 6           bcs   noslow
 7           lda   speed
 8           sta   oldspd
 9           and   #$7F
10           sta   speed
11  noslow   rts
12
13  fastgs   sec
14           jsr   $FE1F
15           bcs   nofast
16           lda   oldspd
17           and   #$80
18           ora   speed
19           sta   speed
20  nofast   rts
21
22  oldspd   ds    1
```

I know Eric taught me a lot with his original joystick routines; I hope you find mine to be as instructional.

Software Evolution

by Steven Lepisto

Sometimes I think software is a process and not a thing. It changes over time as new ideas and techniques are learned and applied. Occasionally there will come along a really novel idea which will revolutionize the whole view of software but those kind of ideas are quite rare. In reality, most changes in software are evolutionary: small changes that are applied to make the software better and better over time.



An early mesozoic hacker (MSDOSus Hackemupus)

For example, back in volume 1 numbers 3 and 4 of the Sourceror's Apprentice, Mr. Lambert published an article by me detailing some code for reading an Apple joystick in a different way. Nothing new here as it had all been done before in one form or another. I simply created one way of doing it and took the opportunity to share this approach with others. Well, after writing that article and code, I have managed to make improvements in how I use the code. I have even managed to make some small improvements in the code itself. All the changes were made in the DOJOYSTICK and UPDATEJOYSTICK routines. The other routines still work just fine and need no changes (as far as I'm concerned at any rate). The changes are as follows:

1) I have modified the DOJOYSTICK routine, which processes the raw data from the joystick, so it no longer handles the keyboard in parallel with the joystick. Instead, I have found it more useful to break out the keyboard handling into a separate routine so it can be called independently of the stick handling. This allowed me to have simultaneous keyboard and joystick handling for a two player game. This approach has also changed how I use DOJOYSTICK.

(All line numbers are in reference to those printed

in S.A. volume 4 number 4.)

To eliminate the parallel keyboard support (and some dead code), replace lines 198 through 214 with

```
    lda     stickstate
```

If you wish to keep the keyboard support, just delete lines 210 through 214 which are unused instructions left over from an earlier version.

Because of the need to process state values from different input devices, DOJOYSTICK now becomes a general processing routine and no longer dedicated to a joystick. This means that two player games with simultaneous input are now much easier to accomplish since all input devices can be treated as a joystick. Also, it is now possible to modify DOJOYSTICK slightly to pass information to and from the routine in the registers for faster and easier use of the routine. These changes are not strictly necessary since you can simply copy the device's state value into STICKSTATE, call DOJOYSTICK and then copy the new value in STICKSTATE back into the variable holding the device's old state value. You must also save or use JOYVECTX, JOYVECTY, BUTTON_STATE, and TRIGGER before the next call to DOJOYSTICK since they are changed each time the routine is called.

To make DOJOYSTICK use registers for passing information (and thereby reduce the number of external variables you need to deal with), make the following changes:

```
a) insert after line 268:

    ldx     joyvectx
    ldy     joyvecty

b) and replace lines 198 through 214 with

    sta     stickstate
```

(if you have made the change to eliminate keyboard support already mentioned, replace the LDA STICKSTATE with the STA STICKSTATE at the beginning of the routine.)

To use DOJOYSTICK now, pass in the A register the state value you want to process. When the routine exits, the A register will contain the new state value, the X register will contain the x vector, and the Y register will contain the y vector. BUTTON_STATE and TRIGGER will still contain those appropriate values. JOYVECTX, JOYVECTY, and STICKSTATE no longer have to be made external to the joystick routines file thus reducing the number of external variables to two.

To support the keyboard as a separate device, call DOKEYSTICK separately (don't forget to add an ENT after the label DOKEYSTICK) then immediately call DOJOYSTICK (DOKEYSTICK returns the state value in the A register). You don't have to worry about retaining button press states from one read to the next because there is no reliable way to tell if a key is being held down and therefore the "button still down" and "button up" states are meaningless.

In UPDATEJOYSTICK, the following changes will fix a bug and make it so that the routine doesn't use STICKSTATE (which gets stepped on in DOJOYSTICK). The first two changes correct the bug.

```
a) change line 291 to read

    bmi     :1b

b) insert at the beginning of line 301 the
label ":1b"

c) rename STICKSTATE in lines 301 and 336
to JOYSTICKSTATE.

d) insert after line 89

joystickstate     ds     1

e) insert after line 101

    sta     joystickstate
```

That's it. The bug had to do with the case when a stick wasn't plugged in. The old button state would have random results from one read to the next meaning that if a stick wasn't plugged in and someone pressed the open-apple key, the routine would return erroneous results for the last button state read which all means that "button up" and "button still down" states would be inaccurate.

So much for changes to the routines. I do have a new

way of using said routines which grew out of many experiments. Specifically, I now call UPDATEJOYSTICK and then immediately call DOJOYSTICK to process the state value thusly:

```
jsr   updatejoystick
txa
jsr   dojoystick
```

(UPDATEJOYSTICK returns the current state value in the X register so a TXA is needed between the calls). I read the stick then process it all at once. This keeps the process of reading and handling the stick (and other devices) all in one area for easier changing. I can now have a single routine for each device that, when called, will update that device's state value and return specific information needed by the calling routine. All nice and neat.

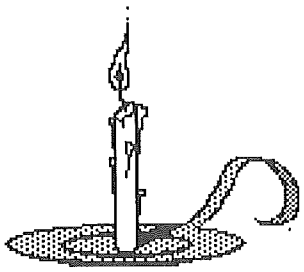
Software is changing all the time. By the time I put out a program, the code has undergone a number of changes from the day of conception. Not only has the design changed but how it has been implemented has undergone significant change. That's why I think software is a process. My joystick routines are one example of that process of change. Another real world example is an animation technique on the IIgs of using stack manipulation on the super hires screen. Alien Mind is the first game I'm aware of that uses a form of this technique. Then came Zany Golf with incredibly fast scrolling of a very large area. Then came Sword of Sodan with its nearly full-screen scrolling with fast animating figures 2/3's the size of the screen. Each game builds on the techniques of the last to create better and better software.

Software changes and improves. This because the programmer him/her -self grows and changes. We experiment with new ways of doing things, having seen that something thought impossible by most is in fact quite possible when someone goes and does it. As we experiment we learn and grow in experience, broadening our scope of understanding which in turn allows us to experiment further, the process feeding itself in a never-ending cycle.

And so Software lives on.

(Editor: Say Steve - any chance you'd enlighten us regarding the graphics techniques you mentioned?)

Some Light on Resources



by Ross W. Lambert, Editor

System disk version 5.0 (and it's progeny, the newly released 5.02) has introduced some brand new concepts for GS programmers. One of the most important and least understood is that of resources.

What they is

A resource is an amazingly protean little beastie that can be all things to all programs. It is a mistake to pigeonhole resources as merely text, icons, dialogs, and menus, although they are those things, too. In reality, a resource is whatever you want it to be. It is *any* data structure your program might need, including tokens for a programming language, parameter settings, and even program code! CODE resources are quite common on Apple's other machine (note that I did not utter the "M" word).

Although resources themselves are new, the concept of positioning a program's data outside the program code proper has a long and glorious history on the Apple II. We all know, for example, that we can change Merlin's behavior by manipulating the PARMS file. Apple also has long espoused placing a program's text all in one place for easy customization and localization (I've written several 8 bit programs that way and it *does* have advantages). And if you've ever BLOADed a binary file into an Applesoft program, you've been using an external resource of sorts.

Resources on the GS, then, are a formalized, easily managed procedure for tucking your program's data away in a nice safe little cubby hole. This cubby hole is often referred to as the "resource fork". The part of a file that holds your program code is referred to as the "data fork", and yes, it is the "normal" part of the file we're used to dealing with. At this point it is important, conceptually, to distinguish between the data your program *uses* (which we'd probably want in the resource fork) and the data your program *generates* (which we'd probably put into the data fork of a separate document file).

The obvious advantage to using the "fork" system is that the resources travel around with the file. If someone copies your program from one disk to another, you don't have to provide a list of all the subprograms, binary files, and parameter file lists that need to be copied with it (ever forget to copy one of AppleWorks' segments?). Instead, due to the magic that is resources, the user just copies your program. All of your resources just travel right along with it.

Another significant advantage to resources is that the Resource Manager does all the dirty work. If you use the new `_StartUpTools` call Jay discussed last month, the Resource Manager will be automatically started and your resource file opened. If you do your tool set startups the old fashioned way, you'll need to explicitly start the Resource Manager and open your resource fork. At any rate, the Resource Manager provides a kind of virtual memory system for resources.

Let's say that you have a large program with a ton of resources operating on a computer with limited memory. If you mark your resources as purgeable, the Resource Manager will pull them into memory as there is room and as they are needed. If you are working on a larger system (memory-wise), all of the resources will stay in memory. *(Note: There are several schools of thought regarding the proper manipulation of resources. Some people promote ditching resources under program control - i.e. forcing a memory compaction - in order to avoid memory fragmentation, others say let the system do it. We'll step into* ***that*** *fray in a few weeks.)*

The key to this functionality is the way the `_LoadResource` call works. If your resource is in memory, it returns a handle to it. If it is not in memory (purged due to memory compaction or whatever), it loads the resource from disk and then returns a handle to it.

If you want to get at the resource data itself, you simply access it via the handle, perhaps locking the block and getting a pointer.

Resource, resource, who's got the resource?

One interesting aspect of resources is that *my* program can open and read *your* program's resources. This is how resource editors work, for example. In fact, a file can have no program code and *only* a resource fork (in truth, the data fork is said to be empty). In fact, during program development, you will probably *want* to have your resources in a separate resource file, i.e. not in your application's resource fork. In such an instance you will need to open the resource file and load the resources yourself.

The reason for working that way is that your resources, once developed, will probably change less often than your program code. Since each recompilation produces a *new* application, you'll be constantly having to read in the resources and stuffing them into the resource fork of the application. 'Tis far easier to read them from an external resource file until yer' done.

How do I get started?

Now that we've delivered the "teaser" and told you all about resources and their usefulness, I must let you down a little. Developing the resources themselves is a fairly involved process, most easily done via a resource editor. Such editors usually create a sort of environment where you "draw and drag" your controls, type in your menu names, or otherwise allow the quick and easy entry of resource data. The editor itself then does the hard work of writing the data to a resource file.

Unfortunately, as Jay pointed out in his article this month, Apple chose to make their initial editor (called Rez), an APW add-on instead of a stand alone application. Jay outlined a process whereby you can still use Rez, but it is a little convoluted and requires APW or Orca.

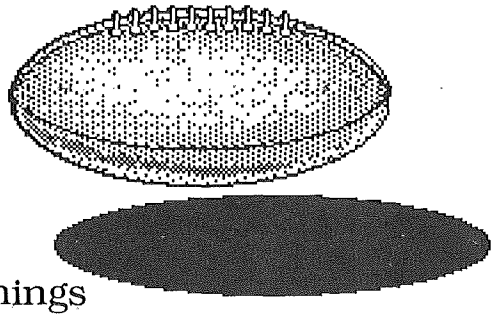
Fortunately, commercial vendors have not been quite so near sighted (actually, I think Apple has a right to support their product - APW - but they'd jolly well better come up with a stand alone application, too.) The ByteWorks (ironically!) is furiously developing a new DesignMaster which will develop resources as well as source code. And another prototyper, Genesis, is also underway. Best of all though, a well placed source at Roger Wagner Publishing (thanks Jeff!) told me that Glen Bredon is also developing a resource editor for us Merlin aficionados. None of the companies involved, however, have even suggested a rough release date. (...a deep sigh of impatience was heard throughout the land...)

Even though it'll be awhile before you can use a resource editor (easily) from Merlin, it is not too early to survey the landscape. Almost any data structure in your code (window definitions, icons, etc.) can be fodder for inclusion in a resource. Look around - I think you'll be amazed.

== Ross ==

More Resource Info

...And Jay Picks Up the Ball



by Jay Jennings

When IIgs System Disk 5.0 came on the scene it brought a new feature with it...resources. This article is for those programmers who know what resources are, but don't know how to implement them.

When I decided it was time to dive into this new area I found that not only was there a lack of sample source to study, but the documentation that comes with the Rez compiler assumes you already know how to use them. I'm sure the information in the binder will be useful at some point in the future, but what I need at the beginning is some sort of tutorial. That's what I hope to do with this article.

There's a big minus to using resources right now. Until Genesys, Design Master, and other resource editors start shipping, the only way to use resources is with the Rez compiler. And that runs under the APW or Orca shell. The Rez compiler is available for about $50 through APDA and you can find the Orca shell (under the name Orca/M assembler) in most mail order ads. Glen Bredon, author of Merlin, is planning on some sort of resource support for Merlin in the future.

Assuming you have the necessary tools, we're now going to put together a program that starts all the tools, throws up an alert window, and then quits. No, it's not very exciting, but it'll get you started in the brave new world of resources!

Let's deal with the StartUp and ShutDown routines first. If you typed in the new Generic StartUp routine in the last issue of SApp, you have almost everything you need. By changing just a few lines in the code we'll tell the program to look for the list of tools it needs in the resource fork. Here's what the tool StartUp call should look like:

```
PushLong #0         ;result space
PushWord UserID     ;ID from the Memory Mgr
pea      2          ;next ref is res ID
PushLong #1          ;ID num of our resource
Tool     $1801      ;StartUpTools call
PullLong SSRec     ;we need this for ShutDown
```

There are only two changes. Our third parameter used to be a zero which meant the fourth parameter would be a pointer to a table of tools. Now it's a two which means the next parameter will be a resource ID. Just in case you were wondering, a one in the third parameter slot means we'd use a handle in the fourth slot.

Now, where did we get this mysterious resource ID from? We made it up! That's right, these are my resources so I'll give them any ID I want. And you'll notice that the ID for the alert window is also a one.

You can use the same ID for different resource types since the StartUp call is only
going to be concerned with a StartUp resource, etc.

Here's the new format for the ShutDown call:

```
PushWord #1          ;reference is a handle
PushLong SSRec       ;give it back the record
Tool     $1901       ;ShutDownTools
```

When we started the tools we pulled a parameter off the stack and saved it in SSRec. Since we're using resources we pulled off a handle instead of a pointer. And that's why our first parameter for the ShutDown is a one instead of a zero.

We're half done with the new StartUp/ShutDown routine. We have the needed code, but the program would crash if that's all you did. We still need to write the actual resources, so go start up APW and get ready for Rez.

The tool table should look very familiar. It's very close to the table we used last month in our assembly code.

Notice that Rez looks an awful lot like C. If you're a die hard assembly fan I'll give you a moment here to collect yourself. Ahhh...

Here's the Rez code you need:

```
#include "types.rez"

resource rToolStartUp (1) /* the resource
ID */
  {
  mode640,
     {/* array TOOLRECS: 18 elements */
     3,0,
     4,0,
     5,0,
     6,0,
     11,0,
     12,0,
     14,0,
     15,0,
     16,0,
     18,0,
     19,0,
     20,0,
     21,0,
     22,0,
     23,0,
     27,0,
     28,0,
     34,0
     }
  };

resource rAlertString (1)   /* resource ID
*/
  {
  "23/I'm using resources!/^Yeah!\0x00"
  };
```

There you go, that's all the Rez code required. Notice that the string used for the AlertWindow is almost exactly like you've been using all along.

Since we've already seen how to change our StartUp code, let's look at the changes needed in our assembly code to take advantage of the AlertWindow resource.

```
pea        0          ;space for result
pea        #%100      ;alertFlags
PushLong   #0        ;address of sub strings
PushLong   #1         ;resource ID
Tool       $590E      ;AlertWindow call
pla
```

Or, if you're using the great macros that come with Merlin, it would look like this:

```
~AlertWindow #%100;#0
 pla
```

It used to be that the first parameter (not counting the space for result) was a zero or one, depending on whether you used a null terminated string or a Pascal type string. That still works if you're not using resources. With resources it's easier if we break that parameter down into
bits. Bits 3-15 must be zero. Bits 1 and 2 tell whether you're using a pointer (00), a handle (01), or a resource (10). Bit 0 indicates the type of string you're using, C string, or null terminated (0), or a Pascal string (1).

The last parameter you push used to be a pointer to the alert string. But with resources you just pass the resource ID number. In this example I've used ID number one.

Okay, time for some standards. I haven't heard if there are already file naming standards in effect, but they couldn't be any better than these. So adopt these and thumb your nose at people who tell you different.

You're going to end up with five different files when you use resources. Two source files, two intermediate files, and one final program file. Here are the suffixes I use to keep everything straight. Your assembly code already has a .S on the end of it so don't mess with it. End your Rez code with .REZ (which even makes sense). When you assemble your Merlin code into an S16 file you need to end up with a file ending with .D which stands for data. That's the part of the code that goes in the data fork. And compile your Rez code into a file with a .R suffix. That's the code that goes into the resource fork. So, this is what you should have:

```
SAMPLE.S        <- Merlin source code
SAMPLE.REZ      <- Rez source code
SAMPLE.D        <- Merlin S16 file
SAMPLE.R        <- Rez S16 file
```

Now we come to the last step. And this is where file number five comes into play. There's a utility called DUPLICATE that comes with the Rez compiler. From the APW shell type the following:

```
DUPLICATE -D SAMPLE.D SAMPLE  <return>
DUPLICATE -R SAMPLE.R SAMPLE  <return>
```

The first line creates a file called SAMPLE and copies SAMPLE.D into the data fork. That's actually your program code. The second line copies SAMPLE.R into the resource fork of the file. You should now have an S16 file called SAMPLE that is ready to run.

Of course, I didn't give you a complete program. But by pasting the StartUp, AlertWindow, and ShutDown routines together (in that order) you can have a sample routine together in just a few minutes.

Wait a minute. We've gone through a zillion steps to do something we could have done in one step with Merlin. But now we can change our startup routine, or the text in the AlertWindow without reassembling our main program. Of course, you'll still have to recompile the Rez code and then copy (DUPLICATE) it into the resource fork of the file.

(Editor: Not necessarily. c.f. my article - just have Rez put the resources into a file with an empty data fork).

The Sourceror's Apprentice

Copyright (C) 1989 by Ariel Publishing Box 398 Pateros, WA 98846 (509) 923-2249 GEnie: R.W.LAMBERT
All Rights Reserved Apple, Apple II, IIgs, BASIC.SYSTEM, and ProDOS are registered trademarks of AppleComputers, Inc.

Subscription prices in US dollars (Canadaand Mexico add $5, non-North Americanorders add $18 per year)
1 year...$28 2 years...$54 Back issues are $3 each (non-USA add $2)

WARRANTY AND LIMITATION OF LIABILITY: I warrant that the information inTheApprentice is correct and somewhat useful to somebody somewhere. Any subscriber may ask for a full refund of theier last subscription payment at any time. At no time shall I or my contributors be held liable for any incidental or consequential damages in excess of the fees paid by a subscriber.

We here at Ariel Publishing freely admit our shortcomings, but nevertheless strive to bring glory to the Lord Jesus Christ.